CHAPTER 15

Programming Transaction-Oriented Applications with Microsoft Transaction Server

In This Chapter

Working with transactions

Understanding how MTS works

Creating and testing MTS components

Windows 2000 contains two new features that are designed to make COM+ a reality in the enterprise: Microsoft Transaction Server (MTS) and Microsoft Message Queue (MSMQ). Together, these technologies represent Microsoft's latest method for moving components from the desktop machine and local network to the Internet with better reliability than ever before. The shortest possible definition of MSMQ is a smart and automated message container (we see how this works later in the chapter), whereas MTS is the insurance that a transaction will complete as anticipated. As the chapter progresses, you'll see that MTS provides a lot more than this basic functionality; we focus, though, on the transactional capability it provides.

According to the latest documentation available from Microsoft at the time of this writing, COM+ is a combination of standard COM components and MTS. However, throughout the course of this chapter, you see that MSMQ is really an

essential part of the equation as well. As far as I'm concerned, to make COM+ a reality, you need COM for the component, MTS to ensure that the results of using the component are safeguarded, and MSMQ to make the entire application mobile. In short, this chapter is about the difference between standard COM and the new COM+ technology that Microsoft is telling everyone about.

Let's look at these two new technologies in a little more detail. MSMQ allows a developer to create applications that react the same — whether or not the user is connected to the LAN. The connected state is what most developers are accustomed to using today — you create a connection to some type of data source, perform some work, then disconnect from it.

Disconnected applications are something new for developers to learn. They act as if there's a connection, even when no connection exists. These "disconnected" applications are the secret to making sure that the application you write today works in a wealth of scenarios, even those that involve users on the road.

Obviously, there has to be a way to ensure that the transactions the user creates get recorded on the server, even if the user has no connection when the transaction is created. That's where MTS comes into play: It works in the background to ensure that only whole transactions get recorded and that incomplete transactions get rolled back to a known good state. The combination of MTS and MSMQ makes it possible to create an entire range of application types you never dreamed possible before.

Even if MTS and MSMQ functionality stopped there, you may want to know how they work so that you can create the latest trend in n-tier applications. However, these two components can be used for a lot more than just COM+ implementations — you can also use them in standalone mode or in combination. For example, you theoretically could use MSMQ to create a new class of smart e-mail applications. MTS can be used for any type of transaction, not just database work. You could use these two products together to create a special kind of e-mail application that works with objects rather than text or other kinds of information. A user on the road could receive interim application updates using an e-mail type connection that gets activated automatically in the background when the user checks into the office. That's the kind of application I'll cover in this chapter — the standalone application that requires the extra data protection that MTS can provide.

According to documentation available at the time of this writing, any MTS applications you created in the past will be treated as COM+ applications with Windows 2000. Obviously, these older applications won't automatically use new features that Windows 2000 has to offer, but at least your current development efforts won't be wasted. You'll be able to move your current MTS projects to the new operating system environment without change.

The first section of this chapter looks at MTS transactions from a theoretical perspective. We'll look at questions like what precisely constitutes a transaction and why they're useful. This section also talks about the various things you need to do to make MTS available for use.

Once you have a better idea of how MTS works, you can tackle the first example application in the second section of this chapter. This is the kind of application that most of you will work with right away. MTS allows you to create components that not only use simpler logic, but can also do more for you with only minimal code additions. In this case, we look at two common tasks. The first is using transactions to ensure that the task gets completed as anticipated. The second is checking the identity of the requestor using MTS role-based security.

The second section of this chapter also shows you how to work with the various user interfaces provided for working with MTS. You have to know how to set up the transaction so that the component works as anticipated. Obviously, this means performing some type of setup at the server as well. We'll look at what you need to do to mark your components as transactional to ensure that they get serviced as needed. In addition, we'll look at a few other developer issues such as when it's better to set security using the Component Services snap-in versus hard coding it within the component itself.

Understanding Transactions

MTS is all about data and object protection and the services required for managing both data and object protection. It really doesn't do much more for you other than that. However, considering the cost of failed data transmissions and the amount of work they can cause, getting data and objects from one point to another safely is essential. MTS uses secured transactions to accomplish its task. Management is accomplished through the use of the Component Services *Microsoft Management Console (MMC)* snap-in with Windows 2000.

According to Microsoft's latest statistics, COM is running on 150 million systems worldwide, some of which have been using COM since it was introduced as OLE in 1991. These systems will benefit greatly from COM+ because this new methodology packages the various COM-related technologies together and makes them easier to work with. You can find out more about COM+ by viewing Microsoft's COM+ Web site at `http://www.microsoft.com/com/tech/complus.asp`.

One of the more interesting downloads at this site, as of this writing, is a presentation that was given at Tech-Ed 99 that contains Microsoft's vision for COM+ now and in the future. You can find out more about the MTS-specific part of the COM+ picture at `http://www.microsoft.com/com/tech/MTS.asp`. You may want to visit this site to get the latest SDKs and MTS-related white papers.

Finally, if you need help with your MTS project, look at the MTS newsgroups starting with microsoft.public.microsoft.transaction.server. In most cases, these newsgroups can help you solve both usage and programming problems you might experience with MTS.

Using transactions is the short definition of MTS; it's time to look at MTS in a little more detail. The following sections describe various theoretical elements behind using MTS as a component deployment strategy. The first section tells you about transactions, what they're about, and what they mean to you as a programmer. In the second section, you explore the relationship between MTS and COM+. You'll find that these two technologies are inextricably entwined. The next section details how MTS does its work from a conceptual perspective. After that, we look at what applications (a method for grouping the components required for one transaction type together) are all about.

Understanding Transactions

The first question you need to answer is, "What is a transaction?" There are a lot of different definitions for this term floating around in the trade press, many of which are contradictory. Let's begin by looking at the one thing that most people agree on: A transaction is a way of packaging data and commands and then ensuring they reach their destination successfully. In other words, when you submit a form to the database by using a transaction, you can be sure that either all of the data in the form will get added to the database or none of it will. In short, a transaction is insurance that your database (or other application) doesn't get erroneous data placed in it even if it means losing the data for a single transaction (which is normally an entire record for database applications).

MTS is the Windows 2000 component that ensures the data or objects (MTS can encapsulate anything, not just data within a transaction) you transfer from one machine to another arrives in good shape and that it be fully executed. Execution may include a variety of things, even though most programmers look at transactions as being data oriented and used only within a database management context. A good way to remember what MTS transactions are all about is the acronym ACID. The following list provides an overview of ACID.

Atomicity	All of the updates required to complete a transaction are grouped together. Either the entire package of updates succeeds and the change becomes durable, or the package fails and all of the updates are removed.
Consistency	The transaction is a correct change of the system state. It preserves the state invariants, which means that you don't get unexpected results from the transaction once completed.
Isolation	Concurrent transactions can't see each other, and the results from one transaction don't affect other transactions running at the same time. Obviously, this is an extension of the principle of encapsulation for objects. Think of a transaction as a form for an object — all of the updates are encapsulated in such a way that the outside world can't see them.

Durability — You can also term this bullet as fault tolerance. A transaction should be able to survive some level of failure. These failures may include obvious problems such as server system failures or unexpected problems such as communication failures.

ACID describes MTS in its perfect state, which is what Microsoft would have you believe is always the case. However, there aren't any foolproof technologies for the PC; you have to play on some level of failure even if you use transactions. It's at this point that some people differ on exactly what a transaction fault tolerance means. For the purposes of this book, a transaction also infers the capability to roll back the addition of data or the execution of commands until the transaction reaches a point of stability — the point at which the last set of operations completed successfully. A transaction never completes until the receiver accepts the data or commands. The second that the data or commands are accepted, the transaction is complete and the event is over.

Of course, transactions are complicated by a number of environmental factors. For example, in a client/server setup, the number of failure points is relatively limited. In the modern n-tier application world, a transaction may involve a number of clients and servers, any of which can fail. MTS views transaction completion as the point at which the original request is completed. If the original request fails, then all intermediate transactions are rolled back. This means that all of the machines involved in the series of transactions required to answer an original request must maintain state information until that original request is completed.

Another environmental factor that complicates matters for today's programmer is the Internet. Consider the fact that this media is both unstable and prone to connection losses. In short, an Internet-enabled application needs to consider transactional failures that aren't caused by the application or any associated components, but by the Internet itself. A connection failure can still cause the transaction to fail even if all of the required application code works as anticipated. MTS helps application programmers handle this kind of failure by maintaining connection information for the programmer. In addition, the use of MSMQ can make the state of the connection a moot point, as we'll see later in the chapter.

Some people also insist that transactions infer some type of data recovery. In other words, the receiver notifies the sender that the data or command isn't accepted for whatever reason. Though error recovery is a very important thing to have, especially when it comes to database managers, you may find that some transaction methodologies don't allow for any form of recovery. If the data is lost, then it's gone. It's up to the sender to ensure that the data actually gets added to the database or that the commands are executed on the server. Transactions under MTS do have a level of data recovery, although it's uncertain how robust that data recovery is and exactly what it protects. In most cases, you may want to be sure that you include some type of data recovery mechanism within your application in addition to whatever Microsoft provides for you when it comes to critical data.

MTS and COM+

As of this writing, Microsoft would have you believe that MTS and COM+ are so closely tied as to make them impossible to understand apart from each other. Actually, it helps to look at the two technologies together, but as separate entities. In fact, if you want the clearest possible view of MTS and COM+ in a nutshell, COM+ is a superset of MTS and standard COM combined. The following sections describe this view of MTS and COM+ in more detail.

Understanding the MTS and COM+ Goals

Microsoft has certain goals when it comes to both MTS and COM+. Up until now, many programmers have avoided COM because they consider it too complex. Creating components is an error-prone and time-consuming process. In fact, Chapter 13 shows you just how complex even a simple component can get from a programming perspective. So, the first major goal of COM+ is to make components both easier to use and create. MTS and MSMQ allow developers to use a simplified programming model, although it still requires some finesse to create any component.

Critics have complained that Windows NT (the predecessor of Windows 2000) doesn't scale very well. A lack of scalability means that you can have a server that works fine with a small load, but the capacity to do work diminishes quickly as load is applied in the form of additional tasks. The same criticism has been leveled at COM. As soon as you add the ability to service more than one request at a time, the complexity of the component increases almost exponentially. COM+ is designed to enhance application component scalability through the use of transactions, multiple servers, and load balancing.

Component technology as a whole is designed to reduce maintenance problems in several ways. The most important consideration for this chapter is that MTS makes it possible for a developer to create components that make applications truly modular. The business logic that's incorporated within a component gets tested only once, instead of within every application. In addition, MTS allows you to enhance the security of the data that's being manipulated by the component, further reducing potential problems.

Finally, one of the main reasons to use MTS is to ensure that the components you create are always available. Placing the components on the server means that the user doesn't have to have a local copy of the component to get some work done. Loading the component on the server makes it available to everyone at the same time. In addition, bug fixes and updates are no longer a chore since only the server needs to be updated instead of every machine on the network.

Putting Components in Context

Part of the difference between COM and COM+ is the idea of context. You can create a component that has no context at all (which is how most components are created today), but that means you really don't have a good idea of how that component is used by the client. The object exists and is used by a client, but the operating system doesn't know much about the object except that it's using some resources. The inability of the operating system to assist with component management means that the developer must incorporate logic within the component to handle a wide variety of failure conditions, many of which are generic to all components.

Placing the component within a context tells the operating system which services this component requires so that the operating system can balance the needs of all components within the system. In addition, the context adds attributes to the component that you normally need to hard code during the programming process. Obviously, one of the major contexts for any component that needs to provide reliable data transfer is a transactional context that defines how transactions are handled given a certain set of default conditions.

At this point, you may think that COM+ components require a lot of additional work to invoke the services that they need. However, the opposite is true. The context for the component defines what services the component requires. Interceptors, which are part of the operating system, look at this context and set the required services up for the component when the client calls it. On return, the interceptors again look at the context and determine what kind of cleanup the component requires since it has finished using these services. In sum, COM+ uses an attribute-based programming model, which differs from the API-based programming model that COM alone used.

The portion of MTS that handles resource management is the *Compensating Resource Manager (CRM)*. The CRM associates an action, abort logic, and complete logic with your component and sets the resources aside required to complete these tasks. For example, you might create a component that processes order forms. The action might be to accept the form from the client, look up the status of the requested items, then return this status information to the client. As you can see, an action consists of one or more objects that execute on one or more servers on behalf of a single client. The abort logic might be to send an error message to the client instead of the order status, whereas the complete logic might be to place the order in the order database (pending client approval). In short, the component always performs the same task, but you can independently change the result of the outcome of that task to meet changing company needs.

You're currently locked into the context and interceptors that Microsoft provides as part of Windows 2000. The eventual goal is to allow the programmer to define the context and which interceptors are used. This, in turn, allows the programmer to define special services in addition to those normally provided by the operating system. Unless you're creating a third-party product, however, you probably don't need this kind of functionality for most applications.

Working with MTS and COM+ Services

Obviously, the goals that Microsoft has set for MTS and COM+ are difficult to obtain without some type of infrastructure. Windows 2000 includes additional COM+-related services that allow you to create and deploy components quickly. Many of these new services are found within MTS and MSMQ (with MTS providing the lion's share of the new services). The result is that you spend more time working with the business logic for your application and less time worried about mundane tasks associated with developing just about any component.

Even though Microsoft stresses the MTS and COM+ connection, there's actually a lot more to this technology than simple transactions. In fact, we look at the majority of these technologies somewhere within this book. Just to summarize what kind of technologies I'm talking about, the following list provides an overview of all of the services that you can find within COM+ 1.0.

- ✦ Server Components (MTS)
- ✦ Transactions (MTS)
- ✦ Security (MTS)
- ✦ Administration (MTS)
- ✦ Load Balancing
- ✦ Queued Components (MSMQ)
- ✦ Events
- ✦ In-Memory Databases

As you can see, at least half of the services that are associated with COM+ also have something to do with MTS. Microsoft has literally merged the COM and MTS development teams in order to produce COM+. So, what you see in Windows 2000 is the work of both of these teams. Let's spend a little more time talking about the MTS-specific contributions to COM+.

Microsoft has worked hard to make writing servers easy. For the most part, writing a server is now like writing any other component. I'll talk about the particulars of writing servers while working with the example component for this chapter.

The MTS portion of COM+ does offer transactional services, as mentioned earlier in the chapter. You can set the transactional attributes within the component itself, or as property values after you install it on the server. However, I didn't mention two important features that MTS provides. The first is AutoComplete, which allows an operation to complete even if the connection is prematurely broken — provided there's enough information to do so. AutoComplete can also be used if your client leaves the component normally and the component simply lacks the code required to complete the transaction itself. Normally, you have to provide a SetComplete( )

method within your component to perform this task. The second is AutoAbort, which means that a transaction is automatically rolled back (or whatever other abort logic you provide) the instant that the connection is lost. Again, you normally implement this logic within the component using the SetAbort( ) method.

Security is a major problem for just about every area of computing today. It's important to protect your data at all times and at every point of processing. MTS allows you to create a secure environment, yet reduces the complexity of doing so. Microsoft uses a role-based security model. What this means is that you assign users to one or more roles that profile their rights to the component. For example, a typical user role might have one set of rights to the component, whereas an administrator role might have other rights that include those of the typical user. Component security is also implemented at the method level now. This means that one method within a component can actually have a higher security setting than other methods associated with the same component.

The rights assigned to a component flow with the context for the component. If you give a user a certain level of access to the component as a whole, then that access is provided at the method level as well unless you specifically assign the user a different set of rights at that level. In addition, rights flow from one object to the next. This is an obvious addition since the user requires access to all of the components needed to complete a particular transaction. Fortunately, you can override this default behavior to ensure that access to sensitive components is regulated properly.

The final MTS contribution is administrative handling of components. In the past, the components would have been managed using some special-purpose utility. Microsoft has decided to use MMC for all administration in Windows 2000. You actually will manage your components using a COM+ administrative snap-in for MMC named Component Services. This management tool is based on the MTS management tool used previously. We look at this tool later in the chapter after creating a simple component for use in the examples.

Using MTS—An Overview

Like any other COM technology, MTS relies on specific component interfaces. In the case of an object that deals with transactions, you need to implement the IObjectContext interface. The whole purpose of this interface is to tell MTS when a transaction has started, when it has completed, and what the results of the transaction are so that MTS knows what to do next. You can also accomplish these tasks using attributes as mentioned in the previous section, so even though you need to implement the IObjectContext interface, you don't have to call on it within your component's code.

The main purpose of many IObjectContext interface calls is to provide success information to MTS. There actually are four different IObjectContext methods that you can call to provide MTS with feedback on the level of success of a particular transaction. In addition, there are three methods used to determine the current context status. The following list provides you with an overview of these methods.

Table 15-1
IObjectContext Interface Method Summary

Method	*Description*	*Result*
SetComplete( )	The transaction completed as originally anticipated.	MTS can mark the transaction as successfully completed. The IobjectContext object is automatically deactivated upon return to the method that first entered the transaction context.
SetComplete( )	The transaction completed as originally anticipated.	MTS can mark the transaction as successfully completed. The IobjectContext object is automatically deactivated upon return to the method that first entered the transaction context.
SetAbort( )	All or part of the transaction has failed.	MTS needs to mark the transaction as a failure and roll back any updates that the component may have made. The IObjectContext object is automatically deactivated upon return to the method that first entered the transaction context.
EnableCommit( )	The transaction isn't complete.	All of the updates in the transaction have been successfully made, so MTS can commit them. Since the transaction isn't complete, the IObjectContext object remains active, even if it returns to the method that first entered the transaction context.

Method	*Description*	*Result*
DisableCommit( )	The transaction isn't complete. In addition, some or all of the requested updates are still pending, incomplete, or failed.	MTS can't commit the updates in their current form. Since the transaction isn't complete, the IObjectContext object remains active, even if it returns to the method that first entered the transaction context.
IsCallerInRole( )	Allows you to determine if the direct caller of your component is in a specified role.	This method is handy for determining how your component should react in a given situation. For example, you can determine whether the calling component is in a typical user or manager role, then act accordingly when processing information.
IsInTransaction( )	Determines if the calling component is executing within a transaction.	You can use this method to determine how to process a request. In addition, it allows you to look for configuration errors. For example, an administrator may configure a component that really requires a transaction to work without one.
IsSecurityEnabled( )	Normally returns true unless the component is running in the client's process.	MTS always uses security to run components. You can use this method to detect client-side component requests that aren't secure (and are therefore potential security breaches).

Transactions are usually surrounded by quite a bit of code, so it may be difficult at times to figure out what the component author is doing if you look at a completed component. In most cases, a transaction consists of five well-defined steps that you always need to implement no matter how complex the rest of the component code.

Try This

1. Create an object context object.
2. Tell MTS that you're beginning the transaction by obtaining an object context using the GetObjectContext( ) method.

3. Perform any work required to complete the transaction. Track all of the updates that you make. Even if the component doesn't deal with a database, it has some type of data transformation work to do. For example, even a computational component accepts raw data as input and provides one or more results as output.
4. Compare the results that the component has obtained with the results that the component was supposed to provide.
5. Provide MTS with feedback. The SetComplete( ) method tells MTS that the transaction was a success. Part of an error-handling routine is to tell MTS that the transaction was a failure using the SetAbort( ) method.

One of the first things that seasoned COM programmers may note about using MTS and COM+ is that you can't choose a remote platform on which to execute your code using the traditional COSERVERINFO data structure that's normally supplied to CoCreateInstanceEx( ). The GetObjectContext( ) call doesn't provide an argument for accepting the COSERVERINFO data structure. So, the question is how MTS determines which remote host to use to service the call. The answer is relatively simple and more flexible than what COM provides — MTS uses a catalog manager to determine the remote execution host. You can configure the catalog used by the catalog manager using either a configuration interface or within the component itself with the MTS catalog interfaces. In short, the programmer can leave the decision of where to execute a component to the administrator who can configure the component using a standard methodology instead of a custom configuration program. Let's take a more in-depth look at the various catalog management features that Windows 2000 provides to the administrator as part of working with the example component.

Visual C++ programmers have access to a number of MTS catalog interfaces. The following list provides you with an overview of the interfaces you need to know to perform both high- and low-level maintenance tasks with the component.

ICatalog	Allows you to connect with specific servers and access the collections they contain. This interface also allows you to retrieve version information for the collection.
ICatalogObject	Provides access to a specific object and its properties. You can use this interface to change values. There are also methods for obtaining the object name and key values.
ICatalogCollection	Provides access to a collection of objects. There are methods for creating, deleting, modifying, and enumerating objects within the collection. You can also use the resulting object to access related collections.

IPackageUtil	Use this interface to import, export, or shut down a package within the Packages collection. Microsoft has replaced the term *package* with *COM+ applications*. If you compare MTS Explorer and the Component Services snap-in, you may notice this change immediately. I'll discuss the exact nature of COM+ applications in the next section of this chapter.
IComponentUtil	Contains a set of methods for importing or installing a component into a collection. You can work with the component by GUID or by name.
IRemoteComponentUtil	Allows you to access components on a remote server by name or GUID.
IRoleAssociationUtil	Use this interface to associate a role with a component. You can access the component by GUID or by name.

Although this list of catalog interfaces is very impressive, it still may not answer the question of why you can't choose a specific server to work with your component. Another part of the COM+ equation is load balancing. Load balancing is taken care of through a combination of a cluster of servers that contains the components you need and a router that determines which server is best able to meet the need given the current network demand and server load. Theoretically, all you should need to do is request services from a specific server and COM+ can automatically choose the best server out of a cluster to answer the request. Once the server is chosen, the router isn't even part of the picture. The load-balancing mechanism provides a direct connection between the client and the server running the component. In short, the need to access a specific server based on load is no longer a real need — all you need is the ability to request services from a "generic" server and let COM+ do the rest.

Understanding COM+ Applications

MTS components are usually loaded on the server as part of a COM+ application. (The term *COM+ application* replaces the term *package* that was used for MTS alone.) In other words, everything that both the client and server need in order to perform a specific transaction type is placed in one location for easy access. A user application can access one or more COM+ applications to provide a full range of services to the end user. When you look at the Component Services MMC snap-in, what you see are four levels of application hierarchy: COM+ application, component, interface, and method. We see how this works in the real world in the examples later in the chapter. However, for now, it's more important to consider a few packaging requirements that help keep your transactions easy to manage and administer. The following sections outline the three most common problems you may experience when working with COM+ applications: fault tolerance, component activation, and security.

Maintaining Fault Tolerance

The first problem that you may encounter when working with COM+ applications is how to maintain fault tolerance. By default, MTS places all of the components in a COM+ application in the same server process. The problem with this approach is that a failure by one component affects all of the other components in the COM+ application. In short, a COM+ application may experience multiple transaction failures because one component experiences problems.

The obvious solution to this problem is to place all of the components in separate server processes. That way, a failure of one component won't necessarily affect the operation of any other component. However, the problem with this approach is that the speed of the transactions slows because each transaction has to cross process boundaries, and server resources get wasted.

It's important to understand that a COM+ application helps keep all of the components required for a particular transaction type together. Applications may require more than one type of transaction to get the job done. As a result, one of the more efficient ways to handle fault tolerance is to group associated components into separate processes along transaction lines. That way, a failed component affects only one of many possible transactions, allowing the application to partially succeed at least. This solution reduces the impact of using multiple processes on execution speed and incurs a minimal penalty on system resources as well.

Activating a Component

The second potential problem is activation. There are three possible places to activate a component: the creator's process, another process on the same computer, and a process on a remote computer.

The safest place to activate the component is another process on the same computer. This approach does incur some performance penalties, but if the component fails, the main application can recover from the failure.

Activating the component within the creator's process is definitely the most efficient way to do things and ensures the highest possible performance (with an equally high risk). Other problems associated with using the creator's process for component activation is that you can't make use of declarative security — an important MTS feature.

In most cases, you won't want to activate the component on another computer for several reasons. The most important reason is that MTS provides remote activation mechanisms using the Remote Computer and Remote Component folders in the MTS Explorer. Considering the likelihood of serious problems, you should never directly activate the component on another machine.

Maintaining Security

The final COM+ application design problem you need to consider is security isolation. Security is a major problem on any network. Maintaining security means being able to identify that the object requesting access to a resource actually has the required credentials to do so. Declarative security allows you to do just that. When you deploy a component in MTS Explorer, you assign it a security role. This security role gets mapped into Windows 2000 group and user security. In short, each component acts as a security checkpoint to ensure that the user has permission to request a specific service. (Fortunately, the user doesn't have to keep entering his or her password during this process — the identity of the user is checked using his or her token. You can find a good description of Windows 2000 security in Chapter 17.)

Creating a Simple MTS Component

Now that we've gotten some of the theory behind working with COM+, MTS, and MSMQ out of the way, let's look at some specifics of a real-world example component. Server components are normally difficult to write because you have to worry about issues such as threading, concurrency, process management, and deployment. In addition, the new n-tier methodology used to implement modern applications is much more complex than the older client/server model. What this means to the developer is that you also have to take the n-tier structure of the company into account when writing a server component.

Microsoft understands that writing a generic component is difficult and writing a server component is nearly impossible, so Microsoft has reduced the complexity of the component creation process for you. The first thing that you can get rid of is the idea that you need to worry about server-specific issues like multiple user access and concurrency. These issues are handled for you by the COM+ services and mainly by MTS. As a result, the new component model looks more like a client application created for desktop use than a server component. You create the component as a single-user, single-thread entity. MTS handles the issues of threading and multiple-user access for you. In addition, it takes care of the process concerns. Of course, you still need to add code to address both transaction and security housekeeping needs. You may find that the amount of code required is relatively small and very easy to understand.

Deployment is also easier because you can use the Component Services MMC snap-in for management purposes. All you need to do is drag the component from its current location on your hard drive and drop it into the location you want it to appear in the MTS display. As an alternative, MTS does provide a wizard that allows you to install the component whether or not it's registered. You'll see how this process works later in the chapter. For right now, all you need to know is that deployment and management are both extremely easy.

The application we'll create in this example performs a very simple set of tasks. In this case, though, the example contains two components instead of one. The first performs simple math functions (add, subtract, divide, and multiply). The second performs some more complex financial functions (simple interest, compound interest, and ordinary annuity). The reason for all of this extra material is so that you can better see how the various MTS displays work. We don't actually look at every bit of code in the chapter, but it's available on the CD-ROM furnished with the book.

The test program looks similar to the one in Chapter 14. It is a little more complex because we have more to test this time. However, the main emphasis of this example is to see MTS in action, so I've kept the examples simple on purpose.

Once the test component and application are working, we'll look at what you need to do to install the component into Component Services. We'll use the wizard wherever possible to keep things simple, and I'll show you some of the configuration options that a developer most likely would be interested in using.

Creating the Component

This section of the chapter concentrates on creating the example component, which actually contains two ATL objects. We've used ATL in several of the examples in the book so far, but like many of the other examples, this one is unique. There are a few things we'll do differently with this ATL example because it resides on the server. For example, there isn't any reason not to make use of the coding ease that MFC can bring into the picture. Unlike the other ATL examples, we don't send this component to the user machine — execution occurs on the server.

Server-based components are also good candidates for holding multiple objects. Unlike a client situation where small size is a distinct advantage, you can save space on the server by combining multiple objects into one component file. Of course, you want to be sure that those objects are related, as they are in the example.

Remember that the objects we create in this example are designed as single-user, single-threaded components. The use of this programming technique greatly reduces the coding complexity of the example. In addition, the specialized MTS objects used in this example can provide additional features normally not found with an object. I'll talk more about these features during the design phase of the component shell. The following sections show you how to create the component. (We'll install it into Component Services later in the chapter.)

CD-ROM Location: Chapt15\DoMath
Program Name: DoMath.DLL
Source Code Modules in Text: DoMath.CPP, SimpleMath.CPP, FinancialMath.CPP

Designing a Program Shell

The first thing we need to do when creating this component is design a component shell. There are a few modifications to note in this process, most notably the inclusion of MFC. Take time to notice the differences that MFC support introduces in this example. You may be surprised at the difference in ATL programming from the other examples in the book once you add in MFC support. The following procedure gets you started.

Try This

1. Start Visual C++, if you haven't done so already.
2. Use the File | New command to display the New dialog box. Click the Projects tab.
3. Highlight the ATL COM AppWizard icon.
4. Type a project name in the Project name field. The example uses **DoMath** as the project name.
5. Click OK. You see the ATL COM AppWizard - step 1 of 1 dialog box.
6. Choose Dynamic Link Library (DLL) as the server type. You also want to check the Allow merging of proxy/stub code, Support MFC, and Support MTS options. I discussed the reason for checking the Allow merging of proxy/stub code option in previous examples—it makes the component easier to work with and more compact. The Support MFC option makes all of the classes that MFC provides available to us as we code. This, in turn, greatly reduces the complexity of coding the components. However, MFC does bring a lot of extra code with it, which means the actual size of your component is larger. The Support MTS option adjusts the build settings for the component. For the most part, this means that the component uses simplified coding techniques and relies on the services that MTS provides to perform complex background tasks.
7. Click Finish. You'll see a New Project Information dialog box like the one shown in Figure 15-1. One of the things you should notice is that the New Project Information dialog box mentions nothing about either the MFC or MTS support added to this project. This is one time when this final check doesn't help you check your settings, making it more important to double-check them before you start coding your project.
8. Click OK to complete the project creation process.

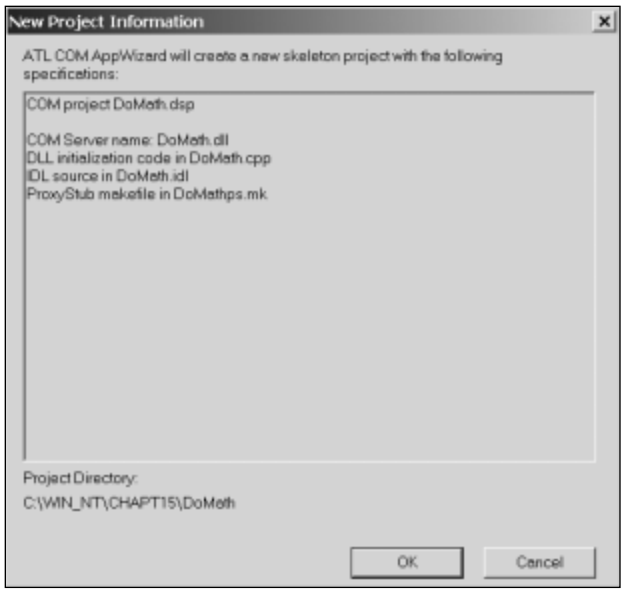

Figure 15-1: The New Project Information dialog box does provide a complete list of project features in this case.

Adding Some Objects to the Program Shell

Let's add two objects to this example instead of one. There are a few reasons for this change in tactic. The most important reason is that we don't transfer this component over the network to the client — all processing takes place on the server. In fact, the MTS support in this component makes it unlikely that you'd ever use this component on the client. For that reason, you can create a single component that contains a lot of related objects to actually save space on the server.

Another reason for placing multiple objects in one component will become clear as we work with the Component Services MMC snap-in later. Packaging like objects together actually makes the task of managing them easier.

Now that we have the reasons for using more than one object, let's add a couple of objects to the program shell we created in the previous section. In this case, let's add two objects. The first contains simple math methods, whereas the second contains financial math methods.

Try This

1. Right-click the DoMath Classes folder in the ClassView tab of the Workspace window and select New ATL Object from the context menu. You'll see an ATL Object Wizard dialog box.

2. Select the Objects option in the Category field. You see a list of simple objects like the ones shown in Figure 15-2. Notice that I've already highlighted the MS Transaction Server Component icon. This is the kind of object that you normally add to a project designed for use with MTS. As I previously mentioned, even though this is an ATL component, it uses specialized settings to make it MTS-friendly.

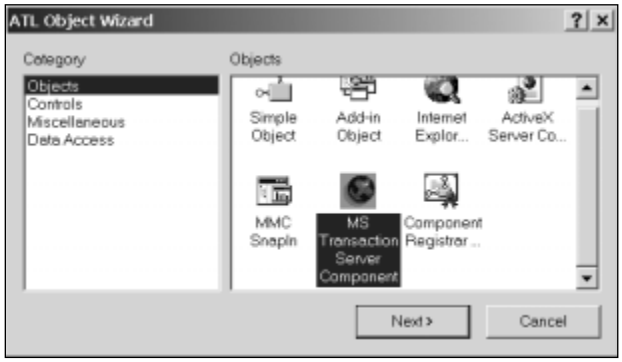

Figure 15-2: The MS Transaction Server Component is designed especially for use with MTS in designing COM+ components.

3. Click Next. The Names tab of the ATL Object Wizard Properties dialog box looks much like the ones that we've viewed in the past. All you need to do is type the name of the object you want to create in the Short Name field. For the first object, I used a short name of **SimpleMath**. For the second object, I used the name **FinancialMath**.

4. Click the MTS tab to see a dialog box like the one shown in Figure 15-3. These are special MTS options that you can choose for the object. In most cases, you want to use the Dual interface option so that your component works with languages that use either the Visual C++ or Visual Basic method for working with interfaces. The only time you need to use a custom interface is if the language you're trying to support doesn't follow the standard methodology for working with interfaces. The IObjectControl interface is optional, but very useful if you want to implement object pooling in your object. Normally, a component gets used by a client, then immediately discarded. Object pooling allows MTS to cache the resource that the object represents for a period of time in the hope that another client will need it. The delay allows MTS to use processor cycles more efficiently, improving overall system performance when an object is called upon at a fairly high rate. In most cases, you only want to implement IObjectControl for objects that clients call upon often since the delay does impose a memory penalty.

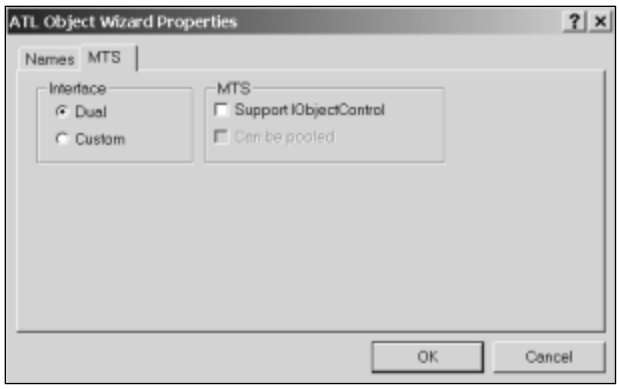

Figure 15-3: The MTS tab contains MTS-specific options that can make your object more efficient.

5. Click both the Support IObjectControl and Can be pooled options for the SimpleMath object since it's likely to get called on a fairly regular basis. The FinancialMath object gets called less often, so leave these two options unchecked for that object.
6. Click OK. The ATL Object Wizard creates the object that you specified.
7. Repeat Steps 1 through 6 to create the second object, FinancialMath.

At this point, we have a program shell and two objects, but no methods (at least none related to the task at hand) within those objects. The following steps help you add the required methods to the two objects. The SimpleMath object contains four methods: Add( ), Subtract( ), Multiply( ), and Divide( ). The FinancialMath object contains three methods: SimpleInterest( ), CompoundInterest( ), and Annuity( ). So, repeat this set of steps a total of seven times; once for each method.

1. Right-click the appropriate interface (ISimpleMath or IFinancialMath), then choose Add Method from the context menu. You should see an Add Method to Interface dialog box. As in all of the other ATL examples in the book so far, this is where you add the definitions for a new method.
2. Type one of the Method Name and Parameters field entries from Table 15-2 into the Add Method to Interface dialog box.
3. Click OK. Visual C++ adds the new method to the appropriate interface.
4. Repeat Steps 1 through 3 until you've entered all seven methods for both interfaces.

Table 15-2
Method Names and Parameters for ISimpleMath and IFinancialMath Interfaces

Method Name	*Associated Interface*	*Parameters*
Add	ISimpleMath	[in] int iValue1, [in] int iValue2, [out, retval] int *iOutput
Subtract	ISimpleMath	[in] int iValue1, [in] int iValue2, [out, retval] int *iOutput
Multiply	ISimpleMath	[in] int iValue1, [in] int iValue2, [out, retval] int *iOutput
Divide	ISimpleMath	[in] int iValue1, [in] int iValue2, [out, retval] int *iOutput
SimpleInterest	IFinancialMath	[in] float fPrinciple, [in] float fRate, [in] int iTime, [out, retval] float *fAmount
CompoundInterest	IFinancialMath	[in] float fPrinciple, [in] float fRate, [in] int iTime, [in] int iCompPerYear, [out, retval] float *fAmount
Annuity	IFinancialMath	[in] float fPayment, [in] float fRate, [in] int iPayments, [out, retval] float *fValue

Adding Some Code

The code for this example isn't designed to show the inner depths of mathematical calculations in the computer world. All it's designed to do is help you get started with MTS and discover how it deals with transactions. With that in mind, Listing 15-1 shows the simple code for the Add( ) method. The code for the other three simple math functions is similar (except for the actual calculation), so I don't show it here for the sake of simplicity. You can view these additional methods in the SimpleMath.CPP file located on the CD-ROM.

Listing 15-1: **Add( ) Method Simple Math Representative Sample**

```
STDMETHODIMP CSimpleMath::Add(int iValue1, int iValue2, int *iOutput)
{
    IObjectContext  *pInstanceContext;  // Object context
    HRESULT         hr;                 // Operation result.
```

Continued

Listing 15-1: *(continued)*

```
    AFX_MANAGE_STATE(AfxGetStaticModuleState())

    // Get an object context.  The object context is used for
    // transactions.
    pInstanceContext = NULL;
    hr = GetObjectContext(&pInstanceContext);

    // If the object context retrival fails, find out why.
    if (FAILED(hr))
    {
        if (hr == E_INVALIDARG)
            AfxMessageBox("Invalid Instance Context Variable");

        if (hr == E_UNEXPECTED)
            AfxMessageBox("An Unexpected Error Occurred");

        if (hr == CONTEXT_E_NOCONTEXT)
            AfxMessageBox("No Context Associated with Object");

        AfxMessageBox("Object Context Not Obtained");
        return 0;
    }

    // Add the two numbers and return them in Output.
    int    iValue = iValue1 + iValue2;
    *iOutput = iValue;

    // Complete the transaction.
    pInstanceContext->SetComplete();

    // Release the object context.
    pInstanceContext->Release();
    pInstanceContext = NULL;

    // Tell the client that everything went OK.
    return S_OK;
}
```

This may look like a lot of code, but it isn't really. In fact, if you take a good look at the example code, the vast majority of it is designed to provide error messages during component execution. These error messages help you locate problem areas in either your code or installation while you work with MTS. There are three sections of operational code in this example.

The first section is added automatically for you. It's the AFX_MANAGE_STATE( ) macro that's used to establish the correct module context for your control. As a result, you should always leave this macro call as the first functional line of code in the method. Since Visual C++ automatically provides this code for you, you don't need to worry about adding it manually in most cases. (Microsoft has written an article entitled, "How to use AFX_MANAGE_STATE in an OLE Control," article ID 127074, that you should read for further details on this particular macro's use.)

The next section is where we work with MTS. There are two steps. The first is to obtain the current object context using the GetObjectContext( ) method. Once you have an object context, you can check its validity using the returned HRESULT value. Now that you have an object context, you can use it to interact with MTS. We see one method of doing so in the code that follows. The last thing we need to do in this transaction is use the object context to tell MTS that the transaction completed successfully using the SetComplete( ) method. Finally, we release the object context using the Release( ) method and setting the variable to NULL.

Unlike most objects, releasing an MTS object and setting its pointer to NULL doesn't automatically free the associated memory. Depending on how you set up the application in MTS, the object is held in memory for some amount of time after you finish with it. The default setting retains the object in memory for three minutes. We see how this works later in the chapter during component installation and configuration. However, it's important to know about this delay because you may run into a linking error that says the component is still in use by the system. In this case, your MTS setup may be causing the problem rather than an error in your code.

The final section of this object is the operational code. This is the same code that we used for the DCOM example in Chapter 14. In short, we simply add two numbers together. Obviously, the example could have added code that detected errors in the addition of the two numbers. If an error was detected, calling SetAbort( ) rather than SetComplete( ) would have told MTS to roll back the transaction to a known good state. Since this is such a simple calculation, the method assumes that it always succeeds.

There are a variety of other ways to use MTS, even in a relatively simple calculation. For example, you may decide that you want to check the security of an individual before allowing them to work with the component. This is actually a two-step process as shown in Listing 15-2. We'll discuss these two steps once you have a chance to look at the code.

Listing 15-2: **Checking Component Security**

```
STDMETHODIMP CFinancialMath::SimpleInterest(float fPrinciple, float fRate, int
iTime, float *fAmount)
{
    IObjectContext  *pInstanceContext;  // Object context
    HRESULT         hr;                 // Operation result.
    BSTR            bstrRole;           // Security role.
    BOOL            lIsRole;            // Check result.

    AFX_MANAGE_STATE(AfxGetStaticModuleState())

    // Get an object context.  The object context is
    // used for transactions.
    pInstanceContext = NULL;
    hr = GetObjectContext(&pInstanceContext);

    // If the object context retrival fails, find out why.
    if (FAILED(hr))
    {
        if (hr == E_INVALIDARG)
            AfxMessageBox("Invalid Instance Context Variable");

        if (hr == E_UNEXPECTED)
            AfxMessageBox("An Unexpected Error Occurred");

        if (hr == CONTEXT_E_NOCONTEXT)
            AfxMessageBox("No Context Associated with Object");

        AfxMessageBox("Object Context Not Obtained");
        return 0;
    }

    // Define a security role.
    bstrRole = SysAllocString(L"Administrator");

    // See if security is enabled.
    if (pInstanceContext->IsSecurityEnabled())
    {
        AfxMessageBox("Security Not Enabled");
        pInstanceContext->SetAbort();
        pInstanceContext->Release();
        pInstanceContext = NULL;
        return 0;
    }

    // Check the security status of the requesting
    // individual.
    hr = pInstanceContext->IsCallerInRole(
            bstrRole,
```

```
            &lIsRole);
    SysFreeString(bstrRole);
    if (FAILED(hr) || !lIsRole)
    {
        if (hr == CONTEXT_E_ROLENOTFOUND)
            AfxMessageBox("Security Role Not Found");

        if (hr == E_INVALIDARG)
            AfxMessageBox("Invalid IsCallerInRole Argument");

        if (hr == E_UNEXPECTED)
            AfxMessageBox("Unexpected IsCallerInRole Error");

        AfxMessageBox("Access Not Allowed");
        pInstanceContext->SetAbort();
        pInstanceContext->Release();
        pInstanceContext = NULL;
        return 0;
    }

    // Add the two numbers and return them in Output.
    float    fValue = fPrinciple * (1 + fRate * iTime);
    *fAmount = fValue;

    // Complete the transaction.
    pInstanceContext->SetComplete();

    // Release the object context.
    pInstanceContext->Release();
    pInstanceContext = NULL;

    // Tell the client that everything went OK.
    return S_OK;
}
```

You need to include MATH.H to get many of the advanced math features that Visual C++ provides to work. This example uses MATH.H for the pow( ) function used in the CompoundInterest( ) and Annuity( ) methods.

As you can see, much of this code is a repeat of Listing 15-1. The biggest difference is that the calculation is a little more complex. However, there's a section of code that allows you to check the users security credentials. Let's look at that code in a little more detail.

The first thing you need to do is allocate a BSTR to hold a security role. You'll see where I got this security role from a little later in the chapter during component installation and configuration. All you need to know for the moment is that this

string has to match the security roles you define for the component precisely. In addition, you need to activate the security role for the component before this feature works as intended.

The next step in the security check process is to ensure that security is even enabled. If it isn't, there isn't any reason to ask if the requesting user is in the defined security role since that role doesn't exist in the current context. Notice the number of steps required to shut down the component if the security requirements aren't met. This is actually a bare minimum and you may have to perform additional cleanup tasks. The message box is for diagnostic purposes; it tells you the reason the component failed to work. Given the nature of MTS components, these diagnostic messages aren't a luxury — you really need them to troubleshoot problems with your component setup. The next thing the code does is use SetAbort( ) to roll the transaction back. It doesn't matter what source of failure occurs in a transaction, you must roll it back if a failure occurs. Finally, the code releases the object context and returns a value of 0. These two steps ensure that your component exits gracefully.

Once we ensure that the component exists in an object context and that security is enabled, it's time to check the user's credentials. The IsCallerInRole( ) method requires two inputs. The first is the role that we want to check. You must pass a BSTR for this call to succeed. The second input is a BOOL variable pointer that holds the user's status on return. A value of true means that the user is in that role; false means that the user isn't in that particular role, but doesn't necessarily mean the user lacks the proper credentials. The distinction between failure to meet a role requirement and having the proper credentials is important. You may place a user in an administrative role, then check for power user credentials in the component. Since administrators are higher in the security hierarchy than power users, they have the proper credentials. However, if you check only for a power user, then the security check may still fail.

Notice that the failure check includes both the BOOL variable lIsRole and the HRESULT value from the call. You need to check both because a security failure may have some other source than an actual security failure. For example, you may have forgotten to define the role during component installation and configuration. If the security check fails, the code still has to go through the proper shutdown procedure to ensure that the component exits gracefully and that no system damage occurs.

Component Installation

At this point, you have a functional component to work with. It's time to install it into MTS and see how components are handled in that environment. You can find Component Services in the Administrative Tools folder found in the Control Panel. Figure 15-4 shows how the Component Services MMC snap-in looks. For those of

you who have used MTS Explorer in the past, the snap-in version should look about the same except for a few changes in nomenclature.

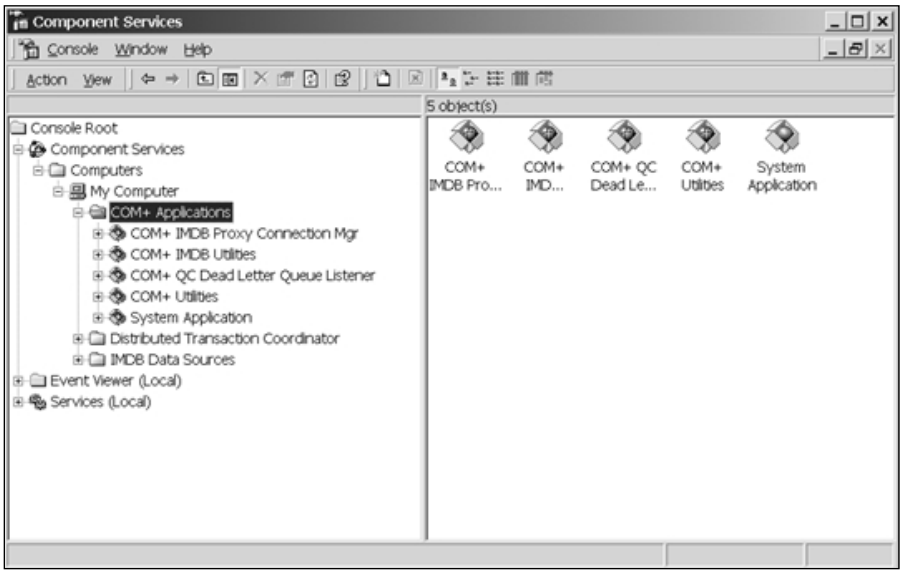

Figure 15-4: The Component Services MMC snap-in is where you work with MTS components.

The first thing we need to do is create a new application for our component. The following procedure shows you how to add a new application to Component Services. I assume that you've already opened the Component Services MMC snap-in.

Try This

1. Highlight the COM+ Applications entry, then select the Action | New | Application command from the toolbar. You'll see a COM Application Install Wizard dialog box.
2. Click Next. You'll see the Install or Create a New Application dialog box shown in Figure 15-5. The Install pre-built application(s) option specifically is designed to allow you to install third-party applications. These applications come with an installation file with an MSI extension that contains all of the particulars about the application. Since we're developing our own application, we need to use the Create an empty application option.

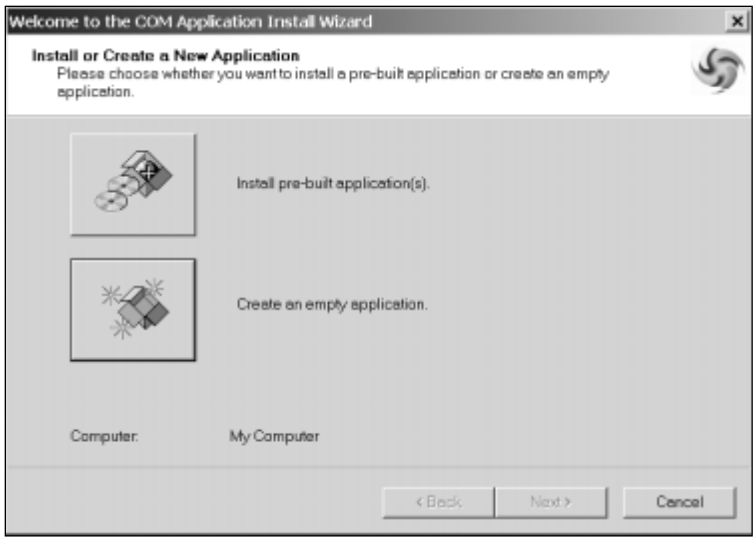
Welcome to the COM Application Install Wizard
Install or Create a New Application
Please choose whether you want to install a pre-built application or create an empty application.
Install pre-built application(s).
Create an empty application.
Computer:
My Computer
< Back
Next >
Cancel

Figure 15-5: The COM Application Install Wizard allows you to create new or install existing applications.

3. Click Create an empty application. You'll see the Create an Empty Application dialog box shown in Figure 15-6. This dialog box allows you to enter a name for your application. It also asks you to decide between a library and a server application type. In most cases, you'll choose the Server application option because it allows the components you create to execute in a separate process. Library applications execute within the creator's process, which means that an errant component can cause the entire application to fail. In addition, library applications don't support load balancing, remote access, or queued components. Library applications do, however, execute faster because there are fewer process boundaries to cross.
4. Type a name for the application. The example uses **Math Aids**, but you can use any name.
5. Choose between the Server application and Library application options. The example application uses the Server application option because it allows more complete testing of the component and better protection during debugging.
6. Click Next. You'll see the Set Application Identity dialog box shown in Figure 15-7. This dialog box allows you to choose the identity of the person used to run the component. You normally choose the Interactive user option because it allows you to test for role-based security using the identity of the person logged into the server. The second option — This user — allows you to set the component up to run as a specific person. It's handy for those situations when you know another server, rather than a user, always call the component. Using this setting is convenient because the component always allows the same level of access no matter which person is logged into the machine.

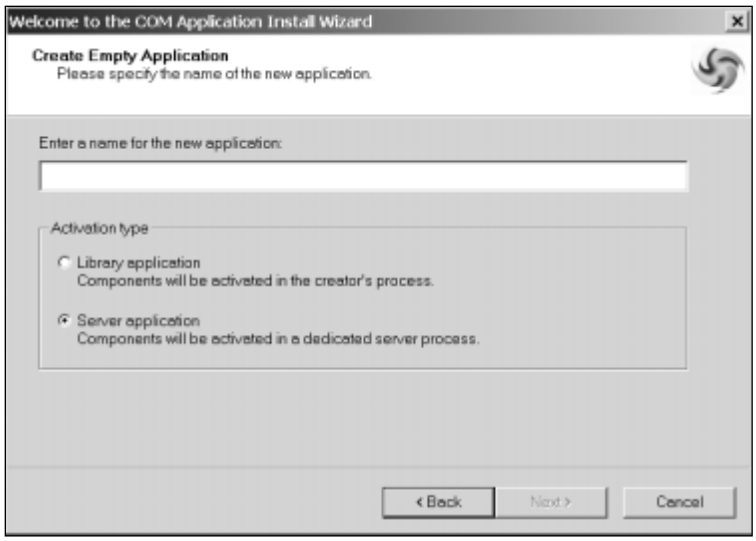
Welcome to the COM Application Install Wizard
Create Empty Application
Please specify the name of the new application.
Enter a name for the new application:
Activation type
Library application
Components will be activated in the creator's process.
Server application
Components will be activated in a dedicated server process.
< Back
Next >
Cancel

Figure 15-6: You need to decide between a library or server application type in this dialog box.

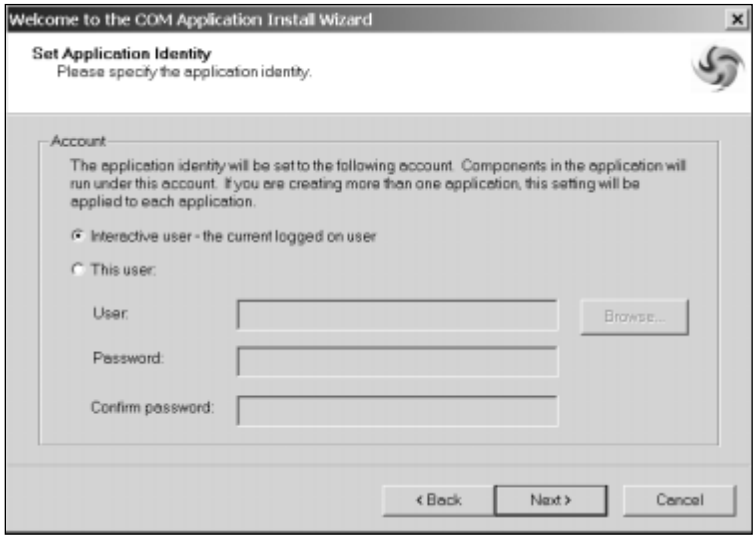
Welcome to the COM Application Install Wizard
Set Application Identity
Please specify the application identity.
Account
The application identity will be set to the following account. Components in the application will run under this account. If you are creating more than one application, this setting will be applied to each application.
Interactive user - the current logged on user
This user:
User:
Browse...
Password:
Confirm password:
< Back
Next >
Cancel

Figure 15-7: Setting the application identity is important because it affects how role-based security is handled.

7. Choose the Interactive user option, then click Next. You'll see a final COM Application Wizard dialog box.
8. Click Finish. The new application appears in the list of applications in the COM+ Applications folder.

The application acts as a container for our component. The component created in the previous section still isn't installed in Component Services. The following steps allow you to install the component, even if it isn't registered on the current machine.

Try This

1. Open the Math Aids application and you'll see two folders: Components and Roles as shown in Figure 15-8. The Components folder holds any components you want to install for the application. The Roles folder allows you to create security roles that you can assign later to the application, an individual component, or a method within the component.

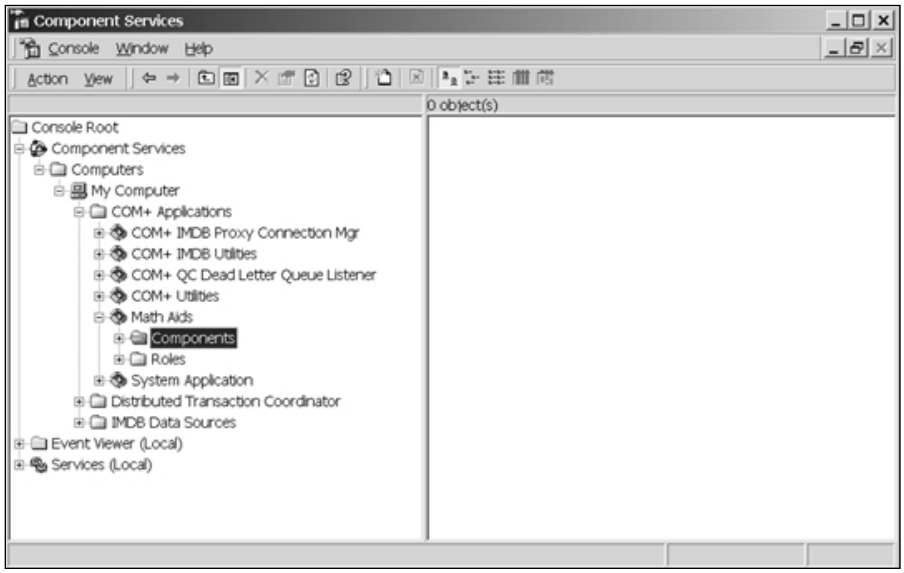

Figure 15-8: All COM+ Applications contain two folders like the ones shown here.

2. Highlight the Components folder; then use the Action | New | Component command to display the COM Component Install Wizard dialog box.
3. Click Next. You'll see the Import or Install a Component dialog box shown in Figure 15-9. The Install new component(s) option allows you to install components that you placed on the server, but haven't registered yet. This is the option that you use when your development platform and the server are

on two different machines. The Import component(s) that are already registered option allows you to add components that have already been registered on the server. This is the option you choose for updates (when the GUID is exactly the same for the new component as it was for the old one) or when the component has been registered for some other reason. For example, you may use your development machine as the test server. Let's look at installing a new component first. If you import a registered component, then proceed to Step 8.

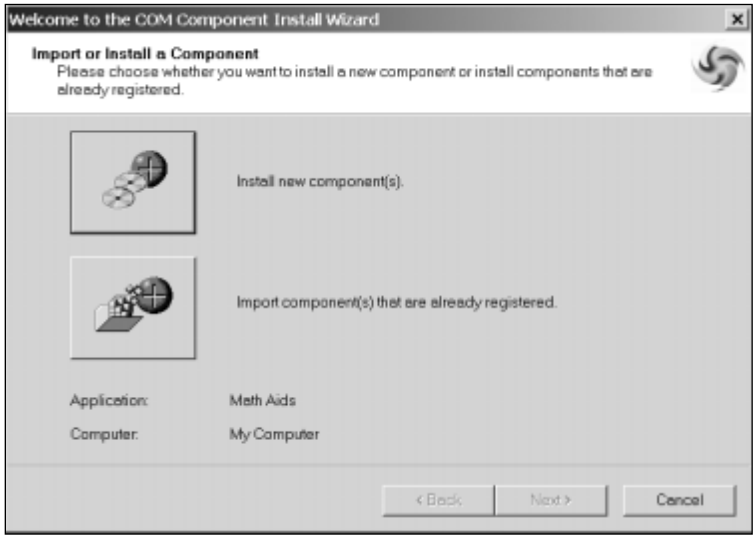
Welcome to the COM Component Install Wizard
Import or Install a Component
Please choose whether you want to install a new component or install components that are already registered.
Install new component(s).
Import component(s) that are already registered.
Application:
Math Aids
Computer:
My Computer
< Back
Next >
Cancel

Figure 15-9: You need to choose between installing a new component or importing an existing component.

4. Click Install new component(s). You'll see a Select Files to Install dialog box (which looks like most of the File Open dialog boxes that you've seen).
5. Find the component file that you want to install. The example uses DoMath.DLL. Highlight the file, then click Open to complete the selection process. You'll see an Install New Components dialog box similar to the one shown in Figure 15-10. Note that the figure shows what you should see when working with the example program. Clicking the Add button, at this point, allows you to add another file to the list (along with any components that the file contains).

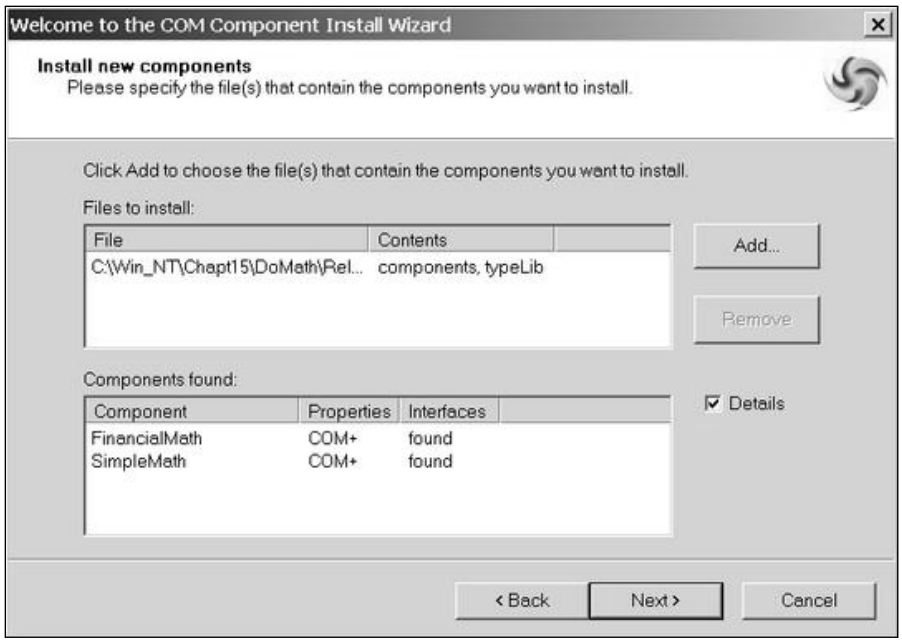

Figure 15-10: This dialog box shows the components contained within the DLL that you've chosen for the application.

6. Click Next. You'll see a final COM Component Install Wizard dialog box.
7. Click Finish to complete the Install new component(s) installation process. You'll see the new components added to the right pane of the Component Services window as shown in Figure 15-11.
8. Click Import component(s) that are already registered. You'll see the Choose Components to Import dialog box shown in Figure 15-12. This dialog box contains a list of all of the components that are installed on the server. If you don't see your components in this list, then you need to click Back and proceed with Step 4 to install the component and register it.

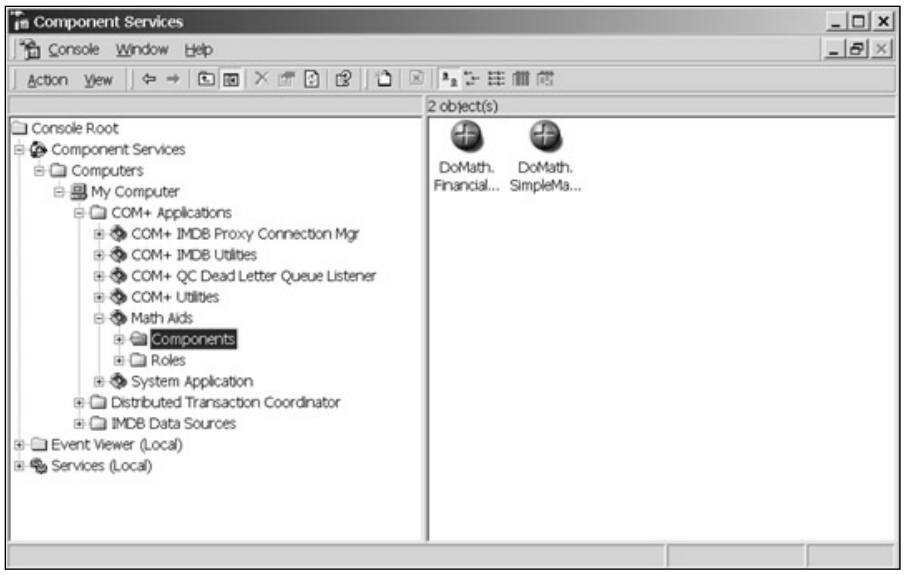

Figure 15-11: The Component Services window contains the new components you've installed once you complete the COM Component Install Wizard.

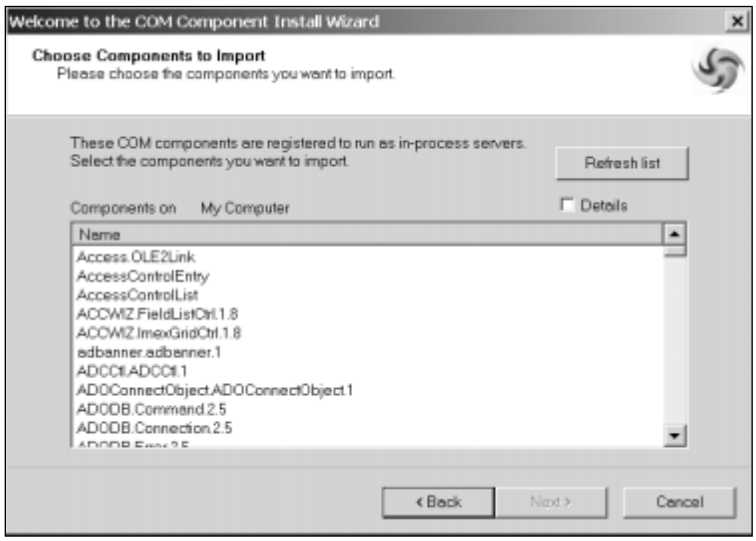

Figure 15-12: The Choose Components to Import dialog box allows you to choose from a list of components that are already registered on the host machine.

9. Locate the DoMath.SimpleMath.1 and DoMath.FinancialMath.1 components in the list and highlight them (depress the Ctrl key while clicking the second component).
10. Click Next. You'll see a final COM Component Install Wizard dialog box.
11. Click Finish to complete the Install new component(s) installation process. You'll see the new components added to the right pane of the Component Services window as shown in Figure 15-11.

Now we have an application and two components installed. However, the components aren't completely operational yet because we haven't installed any security for them. To do that, you need to create at least one, preferably two, roles. Let's call the first role Administrator so that someone logged in as this role can access the financial methods in the FinancialMath component. The second role can be any other value, but for this example let's use User. The following procedure gets you started adding a role to the application.

Try This

1. Highlight the Roles folder, then use the Action | New | Role command to display the Role dialog box shown in Figure 15-13.

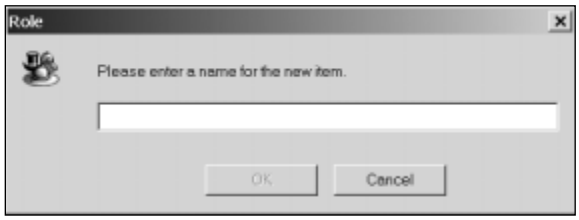

Figure 15-13: The Role dialog box allows you to assign a name to a new role.

2. Type a name for the role, then click OK. For the example, let's use Administrator for one role and User for the second.
3. Open the Administrator (or User) folder, then highlight the Users folder. Use the Action | New | User command to display the Select Users or Groups dialog box shown in Figure 15-14. This is where you choose the users that can access components using the Administrator role.

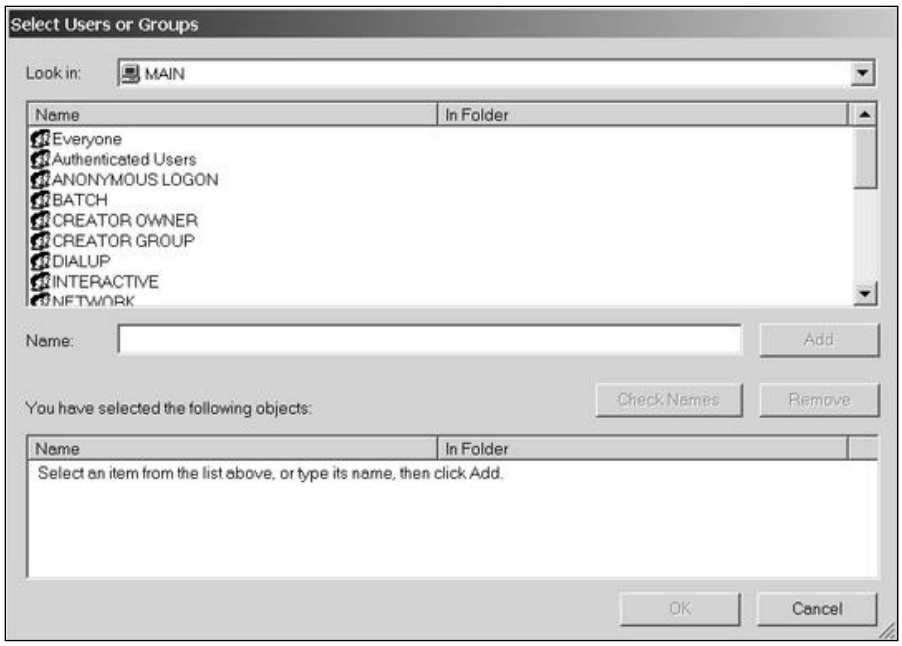

Figure 15-14: The Select Users or Groups dialog box allows you to choose which users can gain access to a component in a certain role.

4. Highlight the Administrator entry, then click Add. You can see the Administrator added to the list of users.
5. Click OK. Administrator gets added to the Users folder in the Component Services window.
6. Repeat Steps 1 through 5 for the User role. However, in this case, use either your own name or a test user name instead of Administrator for the Users folder. Figure 15-15 shows a typical example of the security setup for this example. Obviously, any production components you create will have a more complex security setup that allows users to access the components you create in specific roles.

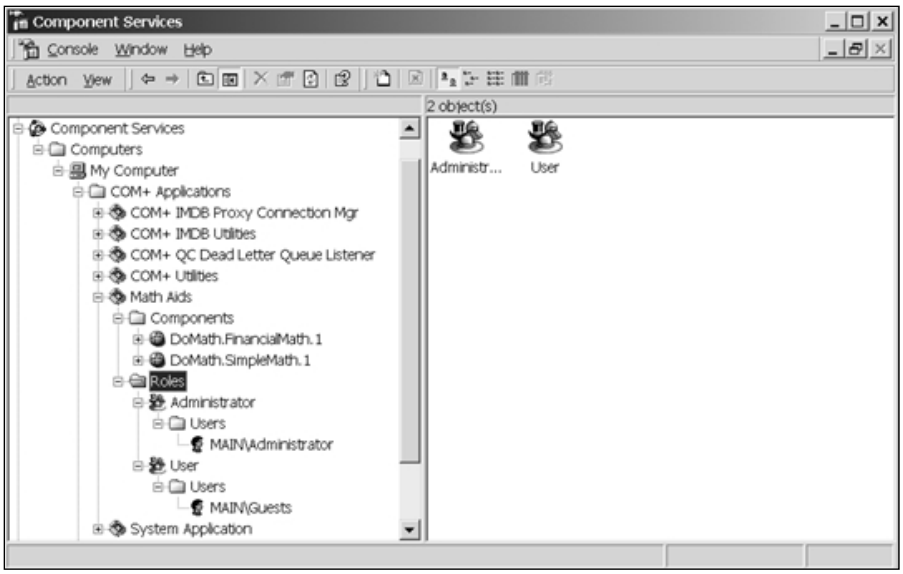

Figure 15-15: The Component Services window shows you the basic security setup for an application.

Security automatically is taken care of for the FinancialMath component because we added it within the source code. However, even if you don't add security while creating the component, you can add it later. Right-click the DoMath.SimpleMath.1 component and choose Properties from the context menu. You see a DoMath.SimpleMath.1 Properties dialog box. Click the Security tab to see a dialog box like the one shown in Figure 15-16. Notice that both of the roles we've created are available for use with this component.

As soon as we add roles to the application, Component Services assumes that we want to use security and enables it for us. The way that the dialog box in Figure 15-16 is set, no one can access the component. What you need to do is check the roles that you want to access the component. For the purposes of this example, check both Administrator and User so that anyone can access the SimpleMath component. Of course, these settings don't affect the FinancialMath component because we've adjusted security at the component level. You also can adjust security at the application and method level by opening the Security tab for either object level.

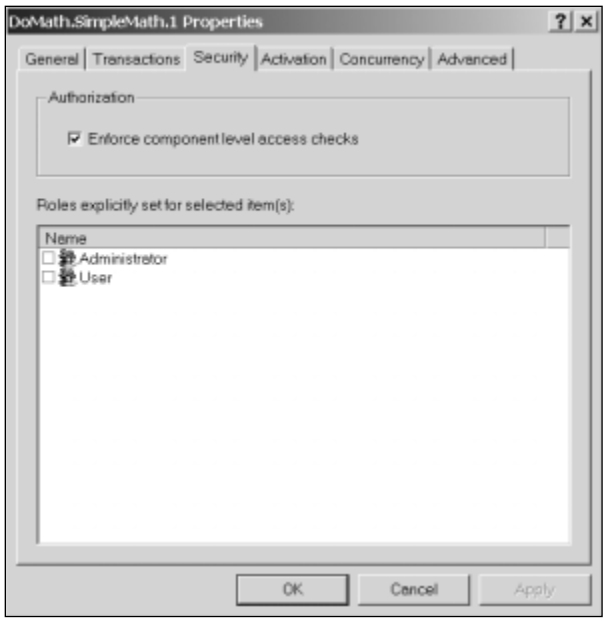

Figure 15-16: Component Services allows you to set security for your component at several different levels.

Creating a Test Application

The component is written, installed, and configured. It's time to create a test application for this example. Obviously, we don't need anything very complicated to test a series of simple math routines. The following sections help you build a very simple test program for the SimpleMath component that we've just installed.

CD-ROM Location: Chapt15\MTSTest
Program Name: MTSTest.EXE
Source Code Modules in Text: MTSTestDlg.CPP, MTSTestDlg.h, MTSTest.RC

There's a more complex version of this example on the CD-ROM in the MTSTest2 folder that includes support for both the SimpleMath and FinancialMath components. I include just a simple test setup in this chapter so that you can see the components at work and how they're displayed within Component Services. The more complex example on disk allows you to explore the financial routines as well as the simple math routines.

Creating the Program Shell

Let's begin by creating the application shell. The following procedure gets you started. (It assumes that you've already completed the basic examples in Chapter 3 and uses an abbreviated procedure.)

Try This

1. Open Visual C++ if you haven't done so already.
2. Use the File | New command to display a New dialog box. Click the Projects tab.
3. Highlight the MFC AppWizard (exe) icon.
4. Type a name for your program in the Project Name field. The sample program uses the name **MTSTest**.
5. Click OK. You'll see the MFC AppWizard step 1 dialog box.
6. Choose the Dialog based option, then click Next. You'll see the MFC AppWizard step 2 of 4 dialog box.
7. Change the dialog title bar text to "MTS Component Access Example."
8. Click Finish. You'll see the New Project Information dialog box shown in Figure 15-17.

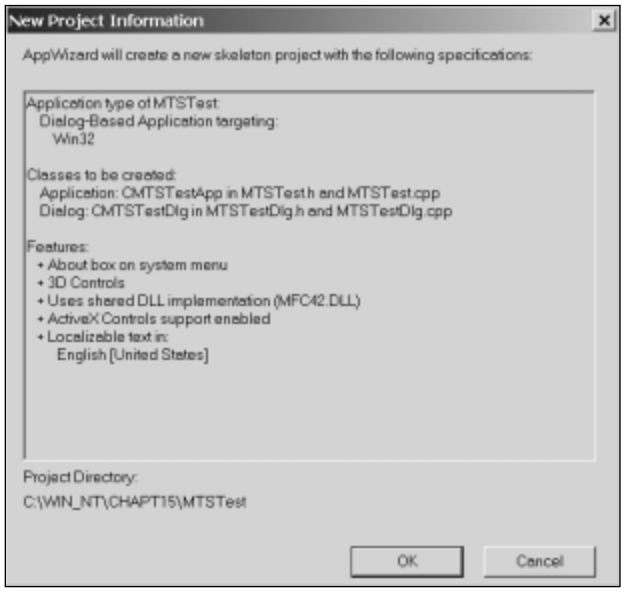

Figure 15-17: The New Project Information dialog box shows that we're creating a dialog-based application with standard features.

9. Click OK. Visual C++ generates your application. You can see a Dialog Editing Window.

Designing the Dialogs

It's time to create a dialog box for testing the SimpleMath component. Figure 15-18 shows the design we'll use for the simple math routines. Notice that there are only three edit controls (two for input and one for results), three static text boxes, and four pushbuttons (one for each math routine type). Table 15-3 tells you how to configure the various controls for use.

Figure 15-18: The first MTS Test dialog box allows us to work with the simple math routines.

Now that you have a dialog box with some controls in it, we need to make those controls accessible to the rest of the application. The following procedure shows you how to create the member variables required for this example.

Table 15-3
MTS Test Simple Math Component Settings

Component	*Property*	*Setting*
IDC_ADD	Caption	Add
IDC_SUB	Caption	Subtract
IDC_MULTIPLY	Caption	Multiply
IDC_DIVIDE	Caption	Divide
IDC_INPUT1	Number	Checked
IDC_INPUT2	Number	Checked
IDC_OUTPUT	Read-Only	Checked

Try This

1. Ctrl+Double-click IDC_INPUT1. You'll see the Add Member Variable dialog box.
2. Type **m_input1** in the Member variable name field.
3. Choose Control in the Category field. Visual C++ automatically chooses CEdit in the Variable type field for you.
4. Click OK. Visual C++ adds the new member variable to your application.
5. Repeat Steps 1 through 4 for IDC_INPUT2 and IDC_OUTPUT. In this case, use **m_input2** and **m_output** as the member variable names.
6. Use the View | Class Wizard command to display the MFC ClassWizard dialog box. Select the Member Variables tab and choose CMTSTestDlg in the Class name field. Your list of member variables should match the ones shown in Figure 15-19. You also need to add functions for the four buttons.
7. Click the Message Maps tab. This is where you add functions that react to the button click event for each of the four math functions.
8. Highlight the IDC_ADD entry in the Object IDs list, then BN_CLICKED in the Messages list.
9. Click Add Function. You should see an Add Member Function dialog box.
10. Click OK to accept the default function name.
11. Perform Steps 8 through 10 for the IDC_DIVIDE, IDC_MULTIPLY, and IDC_SUBTRACT object IDs.
12. Click Edit Code. This closes the MFC ClassWizard dialog box and takes you to the first member function.

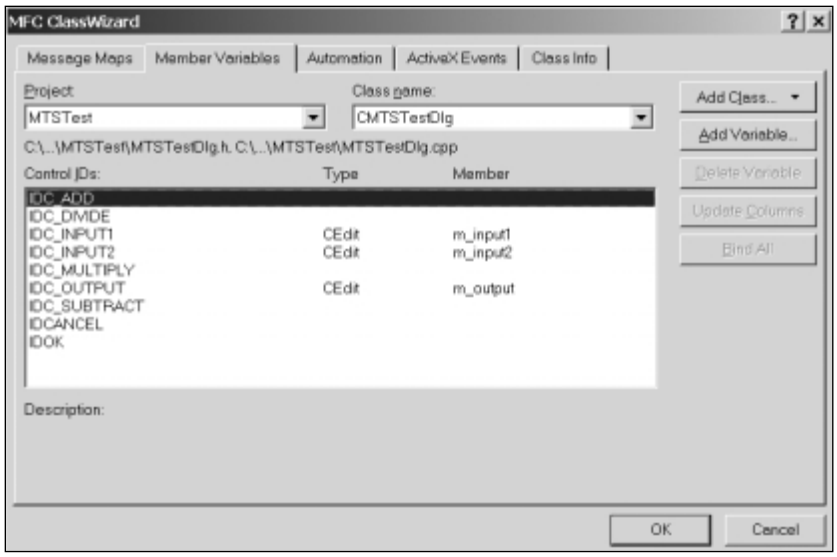

Figure 15-19: The example program requires member variables for each of the edit box components.

Adding Some Code

We're finally ready to add some code. Listing 15-3 contains the code for the OnAdd( ) method. The code for the other three methods is the same. The only difference is the component method that gets called to perform the math and the error message displayed when that calculation fails.

Listing 15-3: **Simple Math Component Test Code**

```
void CMTSTestDlg::OnAdd()
{
    int             iValue = 0;     // Result value.
    int             iInput1;        // First user input.
    int             iInput2;        // Second user input.
    CString         oString;        // Output string.
    HRESULT         hr;             // Object operation result.
    ISimpleMath     *m_pDoMath;     // Interface pointer.

    // Convert the first input value to an integer.
    m_input1.GetWindowText(oString);
    iInput1 = atoi(oString);
```

Continued

Listing 15-3: *(continued)*

```
    // Convert the second input value to an integer.
    m_input2.GetWindowText(oString);
    iInput2 = atoi(oString);

    // Initialize the interface pointer.
    m_pDoMath = NULL;

    // Initialize the COM environment.
    CoInitialize(NULL);

    // Create an instance of the object.
    hr = CoCreateInstance(CLSID_SimpleMath,
                          NULL,
                          CLSCTX_ALL,
                          IID_ISimpleMath,
                          (void**)&m_pDoMath);

    // If the creation failed, exit.
    if (FAILED(hr))
    {
        // Make sure the class was registered.
        if (hr == REGDB_E_CLASSNOTREG)
            AfxMessageBox("Class Not Registered");

        // Make sure we can aggregate the class.
        if (hr == CLASS_E_NOAGGREGATION)
            AfxMessageBox("Class Can't be Aggregated");

        // Display the general failure message and exit.
        AfxMessageBox("Object Creation Failed");
        return;
    }

    // Perform the math calculation.
    hr = m_pDoMath->Add(iInput1, iInput2, &iValue);
    if (FAILED(hr))
        AfxMessageBox("Addition Failed");

    // Convert the result to a string.
    itoa(iValue, oString.GetBuffer(20), 10);
    oString.ReleaseBuffer(-1);

    // Display the output value.
    m_output.SetWindowText(oString);

    // Release and destroy the object.
    m_pDoMath->Release();
    m_pDoMath = NULL;
}
```

If this code looks familiar, it is. This is a slight modification of the code used to test the DCOM component in Chapter 14. The difference is that this example is designed to work with the MTS component. Technically, Microsoft has made it possible to use your existing applications with MTS by making only slight modifications (if any modifications are needed at all).

There are two additional pieces of coding that you need to do. You need to include references to the SimpleMath interface and class descriptions. You need to make two additional entries. The first,

```
#include "..\DoMath\DoMath.h"
```

appears at the top of the MTSTestDlg.H file. The second,

```
#include "..\DoMath\DoMath_i.c"
```

appears at the top of the MTSTestDlg.CPP file. You need both of these entries to make the code work.

Component Testing

It's at this point that we actually get to do something with all of the code that we've been producing so far. Remember that the DCOM example in Chapter 14 acted much as you thought it would. There was no security to speak of and the component worked just as it would on the local machine. In fact, the only major difference was that the code executed on the server rather than the client.

MTS is another step beyond what DCOM provides. You have much more control over how the component acts. More than that, you also have control over how the component transactions get recorded. Let's look at the component in normal operation first. Just open the text application and run it. The four buttons should perform as expected. It's important to make this initial test because we'll change some of the settings as this section progresses.

Now that you know for sure that the components and the test application work as advertised, let's see what happens when you make a configuration change. The first thing you need to do is shut down the application in Component Services. You do this by highlighting the Math Aids folder in Component Services, then using the Action | Shut down command. You can tell if the application has actually shut down by highlighting the COM+ Applications folder. What you'll see is a series of boxes with balls inside of them. If the ball is moving, then the application is running. If you can see the plus sign all of the time, then the application is shut down.

Let's change the security setting for the DoMath.SimpleMath.1 component. As before, right-click the component and choose Properties from the context menu. Click the Security tab, then uncheck both of the roles that we had previously checked. Click OK to make the change permanent. Now, run the test application.

Notice that you get an error message rather than the normal answer. In fact, the process fails no matter which of the four buttons you choose. Once you convince yourself that the component is secure, shut down the application again.

At this point, you might want to reverse the changes to component security that we made earlier, but let's look at method-level security. Open the DoMath.SimpleMath.1/Interfaces/ISimpleMath/Methods folder as shown in Figure 15-20. As you can see, all four of the math routines are easily accessible within Component Services.

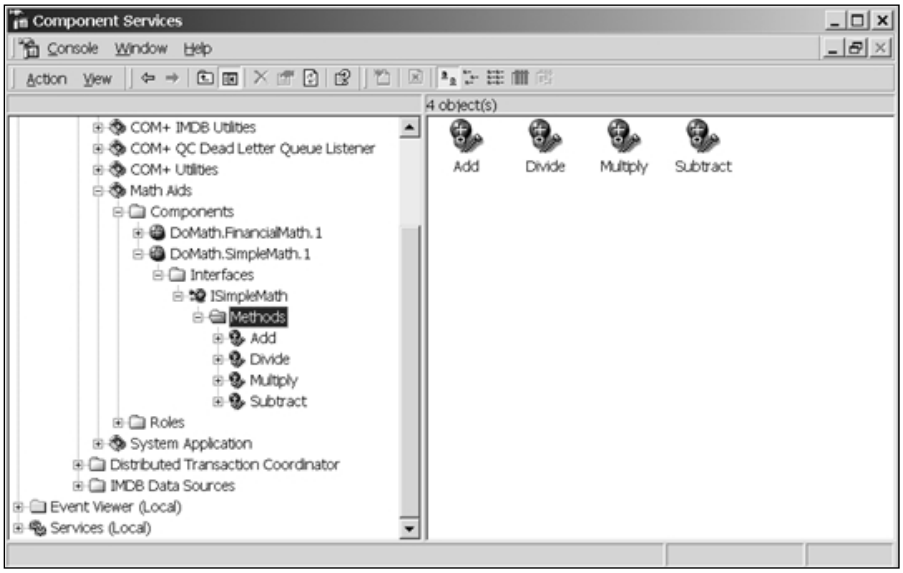

Figure 15-20: You can drill down to the method level within Component Services.

Right-click the Add method, then choose Properties from the context menu. Click the Security tab to see a dialog box like the one shown in Figure 15-21. This is where you set security at the method level. (Remember that you can also set security at the interface level if you want.) Click both of the roles, then click OK. When you run the test application this time, the Add button works just fine, but none of the other buttons work. This is an example of the power of MTS role-based security in action. You control who accesses your components, at what level they access them, and how they're accessed.

It's time to look at another part of the component picture — transactions. Reset the security so that everyone has access to the SimpleMath component at the component level. While you're in the DoMath.SimpleMath.1 Properties dialog box, click the Transactions tab. You'll see a dialog box similar to the one shown in Figure 15-22. Component Services always assumes that a component doesn't support transactions unless you specify that it can. This tab allows you to set the level of transaction support that a component provides.

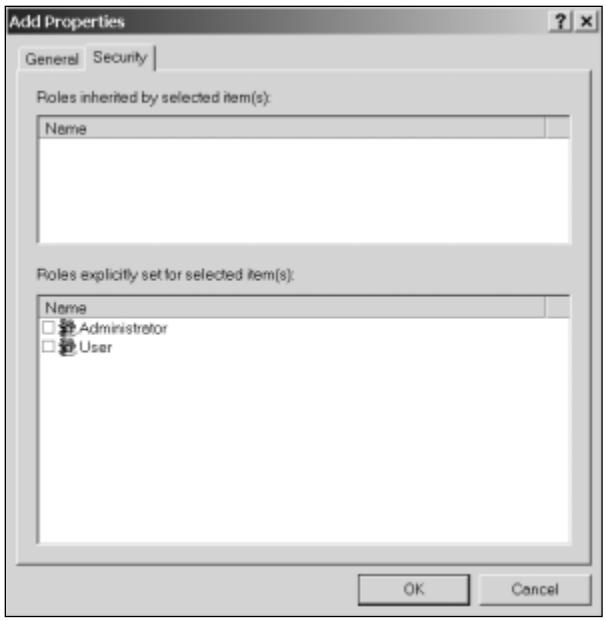

Figure 15-21: Component Services allows you to set security at both the method and interface levels.

Figure 15-22: Transaction support isn't automatic, but it's available.

To make sure that we can record transactions for the SimpleMath component, check the Required option. Now, spend some time working with the test application to generate some transactions. In fact, you may want to open multiple copies of the test application to generate more than one transaction at a time. (Each copy of the test application has its own transaction.) Once you've generated some transactions, open the Distributed Transaction Coordinator/Transaction Statistics folder. You'll see a list of statistics for the component as shown in Figure 15-23.

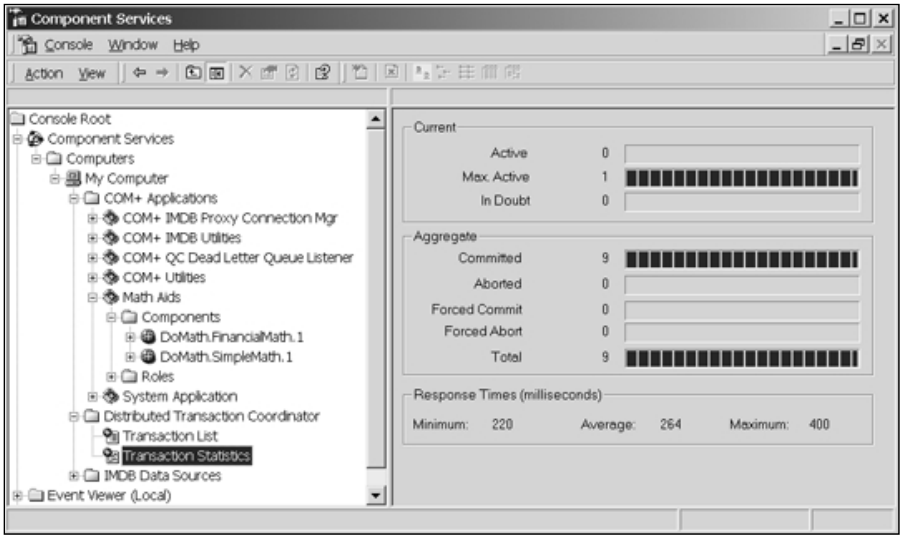

Figure 15-23: Component Services helps you track transactions and their statistics as part of your component management regimen.

There's one other feature we need to look at. Remember all of those security events we were generating as part of the test of the SimpleMath component? If you look at the Event Viewer\Application Log folder, you can see a list of security breach entries similar to the ones shown in Figure 15-24. Double-clicking these entries provides you with additional information about security breaches on your system, including the identity of the person committing the breach. In sum, using Component Services to manage your server-side components makes it much harder for someone to break into your system without being noticed.

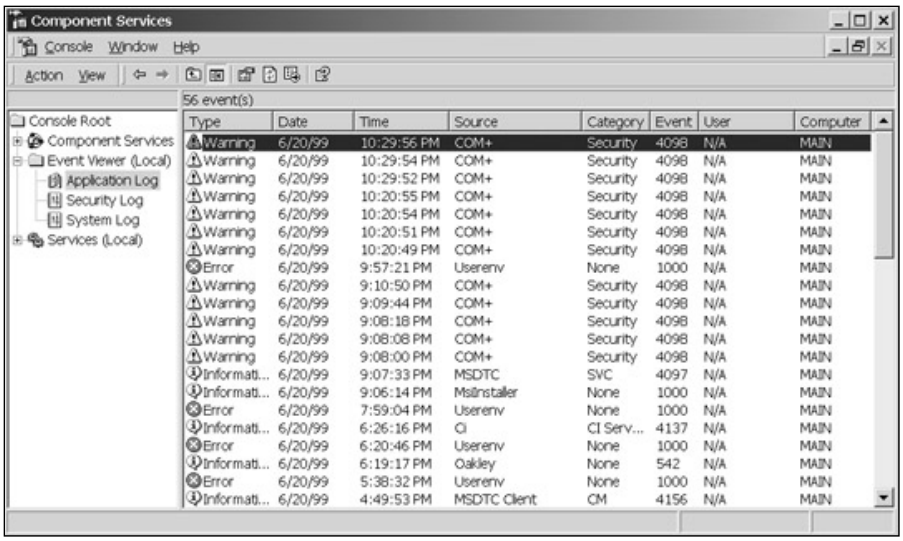
Component Services
Console Window Help
Action View
56 event(s)
Console Root
Component Services
Event Viewer (Local)
Application Log
Security Log
System Log
Services (Local)
Type Date Time Source Category Event User Computer
Warning 6/20/99 10:29:56 PM COM+ Security 4098 N/A MAIN
Warning 6/20/99 10:29:54 PM COM+ Security 4098 N/A MAIN
Warning 6/20/99 10:29:52 PM COM+ Security 4098 N/A MAIN
Warning 6/20/99 10:20:55 PM COM+ Security 4098 N/A MAIN
Warning 6/20/99 10:20:54 PM COM+ Security 4098 N/A MAIN
Warning 6/20/99 10:20:51 PM COM+ Security 4098 N/A MAIN
Warning 6/20/99 10:20:49 PM COM+ Security 4098 N/A MAIN
Error 6/20/99 9:57:21 PM Userenv None 1000 N/A MAIN
Warning 6/20/99 9:10:50 PM COM+ Security 4098 N/A MAIN
Warning 6/20/99 9:09:44 PM COM+ Security 4098 N/A MAIN
Warning 6/20/99 9:08:18 PM COM+ Security 4098 N/A MAIN
Warning 6/20/99 9:08:08 PM COM+ Security 4098 N/A MAIN
Warning 6/20/99 9:08:00 PM COM+ Security 4098 N/A MAIN
Informati... 6/20/99 9:07:33 PM MSDTC SVC 4097 N/A MAIN
Informati... 6/20/99 9:06:14 PM MsiInstaller None 1000 N/A MAIN
Error 6/20/99 7:59:04 PM Userenv None 1000 N/A MAIN
Informati... 6/20/99 6:26:16 PM Ci CI Serv... 4137 N/A MAIN
Error 6/20/99 6:20:46 PM Userenv None 1000 N/A MAIN
Informati... 6/20/99 6:19:17 PM Oakley None 542 N/A MAIN
Error 6/20/99 5:38:32 PM Userenv None 1000 N/A MAIN
Informati... 6/20/99 4:49:53 PM MSDTC Client CM 4156 N/A MAIN

Figure 15-24: The Application Log file helps you detect breaches in security on your system.

Summary

This chapter shows you the theory behind MTS, and how it allows you to create applications that are inherently more reliable than anything you may have created in the past. It also looks at transactions in action from two perspectives: MSMQ and SQL Server. Messaging and database management applications reflect the two places where you'll use MTS in standalone mode most often. Here are some additional conclusions you can draw from the contents of this chapter.

- According to the latest Microsoft documentation, COM+ is a combination of standard COM (including DCOM) and MTS. However, to make COM+ a reality on the corporate desktop, you also need to combine it with MSMQ. These three technologies together allow you to create applications that use components in ways that you could only imagine in the past.
- A transaction is a method of packaging data or commands in such a way that either the whole package is accepted by the receiver or none of the package is accepted.
- The best way to remember what features MTS transactions provide is through the use of the acronym ACID (atomicity, consistency, isolation, and durability).

- COM+ provides eight essential services for creators of components. Of these eight services, four services are provided by MTS. These services include server components, transactions, security, and administration.
- Like all COM technologies, MTS relies on interfaces for communication during a transaction. The interface used with MTS is IObjectContext.
- The IObjectContext interface provides four important methods that a component can use to report the success or failure of a transaction: SetComplete( ), SetAbort( ), EnableCommit( ), and DisableCommit( ).
- The three main problems in determining how to package your application are fault isolation, activation, and security isolation. You need to weigh your options carefully to ensure that the package you create doesn't reduce the effectiveness of using MTS for transactions.
- Component Services allows you to set security for your component at the component, interface, and method levels. You also can use Component Services to monitor security breaches and other potential component problems.
- Component Services maintains a complete log of all of your component statistics, including any transactions.

✦ ✦ ✦